The EvoGen Benchmark Suite for Evolving RDF Data

Marios Meimaris[1,2] and George Papastefanatos[2]

[1] University of Thessaly, Greece
[2] ATHENA Research Center, Greece
m.meimaris@imis.athena-innovation.gr
gpapas@imis.athena-innovation.gr

Abstract. Artificial and synthetic data are widely used for benchmarking and evaluating database, storage and query engines. This is usually performed in static contexts with no evolution in the data. In the context of evolution management, the community lacks systems and tools for benchmarking versioning and change detection approaches. In this paper, we address the generation of synthetic, evolving data represented in the RDF model, and we discuss requirements and parameters that drive this process. Furthermore, we discuss query workloads in the context of evolution. To this end, we present EvoGen, a generator for evolving RDF data, that offers functionality for instance and schema-based evolution, fine-grained change representation between versions as well as custom workload generation.

1 Introduction

The Resource Description Framework[3] (RDF) is a W3C recommendation for representing and publishing datasets in the form of Linked Open Data, a core technology used in the Data Web. The highly distributed and dynamic nature of the Data Web gives rise to constantly evolving datasets curated and managed under no centralized control, with changes found in both the schema and the instance levels. In this context, evolution management - either handled by each individual source during the publishing process or by third-party aggregators during the harvesting and archiving processes - become increasingly important. The significance of systems, frameworks and techniques for evolution management and archiving in the Data Web has been repeatedly pointed out in the literature as a means of addressing quality issues such as provenance tracking, timeline querying, change detection, change analysis, and so on [6,1,21].

Following the proliferation of RDF stores and SPARQL engines, there is a variety of benchmarking efforts, such as the Lehigh University Benchmark[9], the Berlin SPARQL Benchmark (BSBM) Specification [3] and the DBpedia SPARQL Benchmark [18]. Most of them provide real or synthetic datasets of varying size, query workloads and metrics for assessing the performance and functionality of

[3] http://www.w3.org/RDF/

research prototypes or commercial products. Although they offer various parameters for configuring the characteristics of the synthetic data , such as its size and schema complexity, or the type of the generated query workload, their primary goal is to assess the storage efficiency and query performance of RDF systems that operate in a static context.

These issues, however, have not been thoroughly addressed in versioning and evolving contexts, where performance and storage efficiency are greatly affected by evolution-specific parameters. Evolution in RDF data stems from low-level changes (or deltas) in the datasets, i.e., additions and deletions of triples through different time points. These deltas are semantically poor to capture the semantics of the dataset evolution and other parameters come in to play when benchmarking evolution management systems, such as schema vs instance evolution, change complexity, change frequency, data freshness and so on. Hence, any experimentation on versioning and archiving systems must rely on evolution-aware data generators that produce arbitrarily large and complex synthetic data incorporating configurable evolution-specific parameters in the data generation and the query workload production. Two main aspects must be considered towards this goal. First, benchmarking systems must be able to generate synthetic datasets of varying sizes, schema complexity and change granularity, in order to approximate different cases of evolution. Second, benchmarking systems must be configurable in generating representative query workloads with temporal and evolution characteristics.

In this paper, we present, EvoGen, a synthetic Benchmark Suite for evolving RDF that offers synthetic data and workload generation capabilities. EvoGen is based on the widely adopted Lehigh University Benchmark. A preliminary version of EvoGen, [14], addressed generation of successive versions with a configurable shift (i.e. change in size between versions) parameter, without affecting the overall schema of the generated data. We extend the implementation of EvoGen to include configurable schema evolution, change logging and representation between versions, as well as query workload generation functionality. To this end, we build on LUBM's existing benchmark queries, and we provide new ones that address the types of queries commonly performed in evolving settings [17], such as temporal querying, queries on changes, longitudinal queries across versions, etc. EvoGen primarily enables the benchmarking of versioning and archiving RDF systems and change detection and management tools; the provided synthetic workload can also be used for assessing the temporal functionality of traditional RDF engines.

Contributions. The contributions of this paper are summarized as follows:

- we present the requirements and characteristics for generating synthetic versioned RDF,
- we extend the LUBM ontology with 10 new classes and 19 new properties,
- we extend *EvoGen* with configurable schema evolution based on our extended LUBM ontology,

- we implement a change logging mechanism within EvoGen, that produces logs of the changes between consecutive versions following the representational schema of the change ontology described in [22],
- we provide an implementation for adaptive query workload generation, based on the evolutional aspects of the data generation process.

This paper is outlined as follows. Section 2 provides an overview of related work. Section 3 discusses requirements for the benchmark, and Section 4 discusses the parameters of the benchmark in the context of the EvoGen system. Section 5 describes the system's implementation, and section 6 concludes the paper.

2 Related Work

There exists a rich body of literature on RDF and SPARQL benchmarking. Existing works focus on several dimensions, such as datasets, workloads, and use cases [7]. For the dataset and workload dimensions, the requirements for static RDF benchmarks are concerned with providing datasets that are able to represent real-world scenarios and can provide workloads that simulate real-world use cases. For instance, the Berlin SPARQL Benchmark [3] defines two use cases that address different usage scenarios of the data, namely, the Explore use case, which aims at approximating navigational behaviour from customers, and the Business Intelligence use case, that simulates analytical types of queries. Other requirements that authors of benchmarks often cite include quality and quantity metrics in the generated data, such as distinct counts of resources, properties, classes etc., as well as maintaining the selectivity of query patterns between synthetic datasets generated with different tuning parameters (e.g. in [9] and [25]). While most of the existing approaches focus on static benchmarks, the core ideas and motives remain the same when applied to versioned data.

Generally, two types of datasets are considered in benchmarking scenarios; *synthetic data*, which are artifically generated, and *real-world data*, which are taken from existing sources. In this work, we extend the Lehigh University Benchmark (LUBM) [9], a widely adopted benchmark for RDF and OWL datasets. LUBM includes an implementation for generating synthetic data in OWL and DAML formats. Its broader scope includes benchmarking reasoning systems, as well as RDF storage and SPARQL querying engines [28], [11], [12], [2], [4], [10], [20]. It provides an ontology expressed in OWL, in which various relationships between classes exist so that reasoners can perform inferencing. Furthermore, LUBM comes with 14 SPARQL queries with varying sizes of query patterns, ranging from 1 to 6 triple patterns. Because of the fact that these can be limiting when stress testing SPARQL engines, they have been extended in the literature in order to provide more complex patterns (e.g. in [11]). SP^{2}Bench [25] is a generator for RDF data, with the purpose of evaluating SPARQL querying engines. Its scope is mostly query efficiency instead of inferencing, and has been widely adopted in the literature [26,10,13]. Other approaches in the context of RDF and SPARQL benchmarking, such as FedBench [24] and the Berlin SPARQL

Benchmark (BSBM) [3], provide fixed data rather than custom data generation, hence they are not readily capable of providing a benchmark for evolving and otherwise versioned datasets. Nevertheless, the Berlin SPARQL Benchmark does define an update-driven use case, which works on top of the Explore use case. This use case, however, deals with triple additions and deletions that do not affect the schema of the data.

Voigt et al. [29] present real-world datasets associated with query workloads as a means to benchmark RDF engines realistically. Specifically, they draw data from the *New York Times* linked data API[4], which includes data about articles, people, and organizations, *Jamendo* [5], which is an RDF dump of Creative Commons licensed music, *Movie DB*[6], which is a dataset of movies drawn from Wikipedia, Geonames and Freebase, and *YAGO2*[7], a linked knowledge base with data from Wikipedia, Geonames and WordNet. The authors also provide 15 queries for each dataset, and define 5 broad metrics concerning loading time, memory requirements upon loading, performance per query type, success rate for queries, and multi-client support.

The reader is referred to [5] for an extensive study and comparison of RDF benchmarks. Finally, Fernandez et al. [6] discuss a series of metrics for benchmarking archiving systems in Linked Data contexts.

Our approach aims at providing a highly customizable benchmarking suite for creating synthetic and evolving data, with instance-level and schema-level evolution and adaptive query workload generation. For this purpose, we extend LUBM and build on top of EvoGen, an existing synthetic RDF generator. The static component of LUBM is left as-is. For the dynamic (i.e., evolving) data generation, we have implemented tunable functionality where the user can define numbers of versions and percentage of changes between datasets. Furthermore, we extended the original LUBM ontology with 10 new classes and 19 new properties, in order for users to be able to tune schema evolution as well.

3 Requirements

Benchmarking processes adhere to several functional and non-functional requirements for the generation of synthetic data and query workloads, usually determined by specific application use cases in each domain. According to [8], domain-specific benchmarks (in contrast to generic solutions) can provide fine-grained metrics and appropriate datasets for experimenting and assessing the details of a system operating in the context of this domain. In the case of evolving data, there is a multitude of dimensions to address when tailoring the benchmark to custom needs.

[4] http://data.nytimes.com/
[5] http://dbtune.org/jamendo/
[6] https://datahub.io/dataset/linkedmdb
[7] http://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/

3.1 Configurability of the data and change generation process

The benchmark should be able to provide a viable degree of configurability through tunable parameters, regarding the data generation process, the context of the application that will be tested, and the adaptability of the query workload on the specificities of the generated evolving data. The differentiation between benchmarks for evolving settings, and benchmarks for static settings, is that the temporal dimension and the archiving strategy can randomize the data generation process, affecting not only the size but the freshness and change frequency of the data (e.g., a large number of versions is produced with few changes between them), the type and granularity of changes produced between versions and the schema of the generated data. For example, the benchmark must be configurable to the different strategy employed by the evaluated RDF archiving system; a full materialization strategy requires the generation of all versions of a dataset, delta-based strategy requires the generation of one data version (either the first or the most current) and all changes between versions, whereas a hybrid strategy combines these two approaches, requiring a mixed generation of data and changes. For a discussion of different archiving strategies for RDF, the reader is referred to [6,27]. This also implies that a dynamic and adaptive workload is required in order to be consistent with the schema and change information of each generated dataset version.

3.2 Extensibility with evolution-based parameters

As evolving data are by definition dynamic in nature, new requirements are bound to arise as application contexts expand. For this reason, the benchmark is not considered to be exhaustive. Instead, we consider extensibility to be a crucial requirement when designing the parameters of the data generation process and the query workload. For example, there are different approaches for embedding temporal and version information in RDF based on the choices made by the model designer or the capabilities of the RDF store employed; an RDF reification approach uses an extra triple for annotating a resource, whereas a named graph approach uses quadruples to model time and group together resources with the same time or version information. The benchmark datasets and the workloads generated must be easily extensible to accommodate both approaches, or extended to other temporal model alternatives imposed by the application domain.

3.3 Evolution-aware workload generation

For the aforementioned reasons, the workload of the benchmark must be generated adaptively with respect to the required parameters and the generated data. Many traditional benchmarking techniques, with data generation functionality, usually rely on standardized or otherwise fixed query workloads operating on top of the fixed-schema generated data. For instance, the LUBM benchmark that provides the foundations to EvoGen, offers a set of 14 predefined queries that try to address a variety of interesting query patterns with varying complexities. We argue that in evolving and versioning contexts, the fixed queries can only represent static contexts, and it is thus crucial to be able to extend the workload and provide adaptive workloads that reflect the generation process, which is in turn tailored to the user's custom needs. For example, the number of versions and the variations, as well as the complexity of changes between the versions, leads to significantly different outcomes that can impact the same set of benchmarking tests in varying and possibly unpredictable ways.

4 EvoGen Characteristics

4.1 Synthetic data description

EvoGen is based on the prototype implementation presented in [14], which served as a first attempt for a synthetic RDF data generator over evolving contexts. It is based on the widely used LUBM generator, which uses an ontology of concepts drawn from the world of academia. Specifically, LUBM creates a configurable number of university entities, which are split in departments. Furthermore, LUBM generates entities that describe university staff and students, research groups, and publications. Most of these classes are provided in different types of specializations, as defined in the LUBM schema ontology. For example, the generator creates varying numbers of lecturers, full professors, associate professors and assistant professors, as well as undergraduate and postgraduate students. The created entities are interrelated via direct (e.g., a professor can be an advisor of a student) or indirect properties (e.g., professors and students can be co-authors in publications), and their cardinalities adhere to relative ranges that are hard-coded in the generator. LUBM heavily relies on randomization over these types of associations, however, it is guaranteed that the schema will be populated relatively evenly across different runs.

While in [14] we do not extend the original schema, in this work we provide 10 new classes and 19 new properties. The new classes are both specializations (subclasses) of existing ones (e.g. visiting professor, conference publication), and novel concepts in the ontology (e.g., research project, scientific event). Through this extension we are able to implement schema evolution which was not supported in the original version, and at the same time keep the original LUBM schema intact to allow backwards compatibility with existing approaches.

Finally, the current version of EvoGen follows the DIACHRON model[16], a named graph approach, for annotating datasets with temporal information. According to this, time is represented at the granularity of the dataset; all dataset resources refer to the same time point and separate named graphs are used for grouping resources in dataset versions. EvoGen, however, can be easily modified to accommodate other temporal modellings for data generation. Also, time is represented in a ordinal manner in which the time validity of the dataset versions is denoted by natural numbers. Again, other temporal representation such as absolute time or time intervals can be used.

```
ex:change1 rdf:type co:Add_Type_Class ;
co:atc_p1 lubm:VisitingProfessor .
ex:change2 rdf:type co:Add_Super_Class ;
co:asc_p1 lubm:VisitingProfessor ;
co:asc_p2 lubm:Professor .
ex:change3 rdf:type co:Add_Property_Instance ;
co:api_p1 AssociateProfessor13 ;
co:api_p2 lubm:doctoralDegreeFrom ;
co:api_p3 University609 .
```

Listing 1: Example RDF in the change log

4.2 Change Generation

We design and implement a component for semantic change generation, which relies on the change representation scheme presented in [22]. Changes are represented as entities of the Change Ontology, which is able to capture both high level changes, such as adding a superclass, and low level changes, such as triple insertions and deletions. The Change Ontology has been adopted by the community and used in change detection and change representation [22] and in [23] for designing and representing multi-level changes. Also, it is tightly integrated with the temporal query language DIACHRON QL [17]. EvoGen optionally creates a change set between two consecutive versions, that includes all changes between the versions, both on the instance and on the schema level.

4.3 EvoGen Parameters

We follow the approach established in our previous work [14], where we drive the generation process through a set of abstract parameters that reflect the user's needs with respect to the type and amount of changes. Specifically, we reuse the notions of shift, monotonicity and strictness as high level characteristics of the generation process, and we define an extra parameter for class-centric schema evolution. In what follows, we describe these notions.

Parameters regarding instance evolution. Following the definitions provided [17,15], we treat evolution on the dataset level by default. In this context, a dataset D is *diachronic*, when it provides a time-agnostic representation of its content. The instantiation of a diachronic dataset at a given time point t_i denotes the annotation of the dataset's contents with temporal information regarding this time point. Given this, let D be a diachronic dataset, and $D_i \dots D_{i+n}$ a set of dataset instantiations at time points $t_i \dots t_{i+n}$. Then, the *shift* of dataset D between t_i and t_{i+n}, denoted as $h(D)|_{t_i}^{t_{i+n}}$, is defined as the ratio of change in the size of the instantiations $D_i \dots D_{i+n}$ of D.

$$h(D)|_{t_i}^{t_{i+n}} = \frac{|D_{i+n}| - |D_i|}{|D_i|} \tag{1}$$

The *shift* parameter shows how a dataset evolves with respect to its size, i.e., the number of resources contained in each version. Its directionality is captured by signed values, i.e., a *positive* shift points to the generation of versions with *increasing* size, whereas a *negative* shift points to versions with *decreasing* size. It essentially captures the *relative* difference of additions and deletions between two fixed time points, and as a parameter it allows for generating increasingly larger or decreasingly smaller versions through the generation process. In this version of EvoGen, given an input shift $h(D)|_{t_i}^{t_{i+n}}$, the changes are evenly distributed between all versions $D_i \dots D_{i+n}$. This is a limitation of the current version of EvoGen, but will be extended in the future.

The *monotonicity* of a dataset D determines whether a positive or negative shift changes D monotonically in a given time period $[t_i, t_j]$. A monotonic shift denotes that additions and deletions do not coexist within the same time period. Note that monotonicity is not necessarily an aspect of evolving datasets. However, it can be invoked by the user in order to simulate datasets that are strictly increasing or decreasing in size, such as sensory data and historical data.

Therefore, the set of triples that occur in a series of consecutive versions of D between t_i and t_j will be strictly increasing for a monotonic positive shift, and strictly decreasing in a monotonic negative shift. In order to make the ratio of low-level increasing (i.e., triple insertions) to decreasing (i.e., triple deletions) changes quantifiable, we use the notion of *monotonicity rate*, denoted as $m(D)|_{t_i}^{t_{i+n}}$, as a parameter between 0 and 1:

$$m(D)|_{t_i}^{t_{i+n}} = \frac{|t_a|_i^{i+n}}{|t_a|_i^{i+n} + |t_d|_i^{i+n}} \tag{2}$$

where $|t_a|_k^l$ and $|t_d|_k^l$ the number of added and deleted triples between time points t_k and t_l. Formally, we define a dataset D to be *monotonically increasing* when:

$$h(D)|_{t_k}^{t_l} > 0 \quad \text{and} \quad m(D)|_{t_k}^{t_l} = 1$$

, or more intuitively, when the shift is positive and there are no triple deletions between t_k and t_l. In a similar way, we define a dataset to be *monotonically decreasing* when

$$h(D)|_{t_k}^{t_l} < 0 \quad \text{and} \quad m(D)|_{t_k}^{t_l} = 0$$

, or more intuitively, when the shift is negative and there are no triple additions between t_k and t_l.

Parameters regarding schema evolution. The *ontology evolution* parameter of a dataset represents the change on the ontology (i.e., schema) level, based on the change in the number of total classes in the schema. It can be used in conjunction with the *schema variation* parameter that will be defined in what

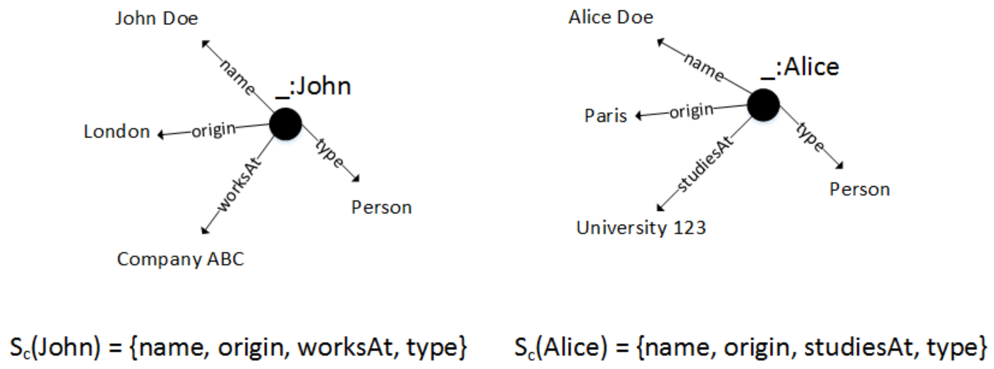

Fig. 1: Two resources of type Person with different characteristic sets. The characteristic sets are shown at the bottom.

follows. The *ontology evolution* parameter, denoted as $e(D)|_{t_k}^{t_l}$, is the ratio of new classes to the total number of classes in t_l:

$$e(D)|_{t_i}^{t_{i+n}} = \frac{|c_{i+n}| - |c_i|}{|c_i|} \tag{3}$$

where $|c_i|$ is the total number of ontology classes at time t_i.

Next, we define the *schema variation* parameter, based on our former notion of strictness presented in [14]. Schema variation property, denoted as $v(D)|_{t_k}^{t_l}$, captures the different schema variations that a dataset D exhibits through time. Because of the schema looseness typically associated with RDF, we recall the notion of Characteristic Sets [19] as the basis for $v(D)$. A characteristic set of a subject node s is essentially the collection of properties p that appear in triples with s as subject. Given an RDF dataset D, and a subject s, the Characteristic Set $S_c(s)$ of s is:

$$S_c(s) = \{p \mid \exists o : (s, p, o) \in D\}$$

and the set of all S_c for a dataset D at time t_i is:

$$S_c(D) = \{S_c(s) \mid \exists p, o : (s, p, o) \in D\}$$

The total number of combinations of the properties associated with a given class gives a maximum number of $2^n - 1$ characteristic sets associated with that class. This is shown in the example of Figure 1, where two instances of the professor class correspond to two different characteristic sets; the first instance has the *name*, *origin*, *worksAt* and *type* properties, and the second instance does not have a *worksAt* property, but has a *studiesAt* property. Given this, we consider $v(D)|_{t_k}^{t_l}$ to be a constant parameter between 0 and 1 that quantifies the percentage of different characteristic sets, with respect to the total number of possible characteristic sets, that the generator will generate in the evolving process. Therefore, for all classes $|c|$ of a dataset D, the percentage of characteristic sets for a given time period is given by the following:

$$E(D)|_{t_k}^{t_l} = v(D)|_{t_k}^{t_l} \times \sum_{i=1}^{|c|} 2^i - 1 \tag{4}$$

We call E the *schema evolution* parameter. In essence, (4) quantifies the number and quality of schema changes in the dataset as time passes.

Parameters regarding query workload generation. EvoGen generates a query workload that is based on six query types associated with evolving data defined in [17]. We briefly provide an overview of the query types and the generated workload in the following:

1. Retrieval of a diachronic dataset. This type of query is used to retrieve all information associated with a particular diachronic dataset, for all of its instantiations. It is a workload-heavy CONSTRUCT query that either retrieves already fully materialized versions, or has to reconstruct past versions based on the associated changes.
2. Retrieval of a specific version. This is a specialization of the previous type, focusing on a specific (past) version of a dataset. The generator has to be aware of the context of the process, and create a query that refers to an existing past version.
3. Snapshot queries on the data. For this type of query, we use the original 14 LUBM queries and wrap them with a named graph associated with a generated version.
4. Longitudinal (temporal) queries. These queries retrieve the timeline of particular subgraphs, through a subset of past versions. For this reason, we use the 14 LUBM queries and wrap them with variables that take values from particular version ranges, and we order by ascending version in order to provide a valid timeline.
5. Queries on changes. This type of querying is associated with the high level changes that are logged in the change set between two successive versions. We provide a set of simple change queries that provide the ability to benchmark implementations that extract and store changes between RDF dataset versions, represented in the change ontology model.
6. Mixed queries. These queries use sub-queries from a mixture of the rest of the query types, and provide a way to test implementations that store changes alongside with the data and its past instantiations.

Parameters regarding the type of the archiving strategy. Finally, we provide some degree of configurability with respect to EvoGen's serialized output. More specifically, we allow for the user to request fully materialized versions, or the full materialization of the first version followed by a series of deltas. This allows using the generated data in scenarios where the archiving process uses different archiving strategies, such as full materialization of datasets, delta-based storage, and hybrid storage, which is a combination of the two.

5 Implementation

We build on the original EvoGen implementation[8], a prototype generator for evolving RDF data with configurable parameters. Specifically, we extended the configurability of EvoGen by implementing change logging, schema evolution, and query workload generation offered in different types of archiving policies, as discussed in section 4.

The system extends the Lehigh University Benchmark (LUBM) generator, a Java based synthetic data generator. In this version, LUBM's schema is extended to include 10 new classes and 19 new properties, which served as a basis for implementing schema evolution functionality. Specifically, we have implemented schema evolution on top of the original ontology, without affecting the original ontology's structure for backwards compatibility.

The high-level architecture of EvoGen can be seen in Figure 2. The implemented functionality includes instance-level monotonic shifts, as well as schema-level evolution by class-centric generation of characteristic sets, as defined in section 4. The parameters that can be provided as input by the user in EvoGen are as follows:

1. number of versions: integer denoting the total number of consecutive versions. The number of versions needs to be larger than 1 for evolving data generation, else the original LUBM generator is triggered.
2. shift: the value of shift as defined in equation (1) of section 4, i.e., $h(D)|_{t_i}^{t_j}$, for a time range $[t_i, t_j]$, represents the percentage of change in size (measured as triples) between versions D_i and D_j. Currently, EvoGen generates monotonically incremental and decremental shifts between consecutive versions, and distributes the changes between all pairs of consecutive versions.
3. monotonicity: A boolean denoting the existence of monotonicity in the shift, or lack thereof.
4. ontology evolution: this parameter denotes the change in ontology classes with respect to the original ontology of LUBM, as given by equation (3).
5. schema variation: this parameter is used to quantify the total number of permuted characteristic sets that will be created for each new class introduced in the schema, as defined by equation (4) in section 4.

The *Version Management* component and the *Change Creation* component are the main components that deal with translating the input parameters to actual instance/schema cardinalities and weights. They compute how many new instances have to be created or how many existing instances have to be deleted for each class of the LUBM ontology, without affecting the structure of the data and the distribution of instances per class.

The functionality is exposed through a Java API that can be invoked by importing EvoGen's libraries into third party projects.

The actual distribution of triple insertions and deletions is performed dynamically in a process that takes into account session information on the evolution

[8] Source code is available at: https://github.com/mmeimaris/EvoGen

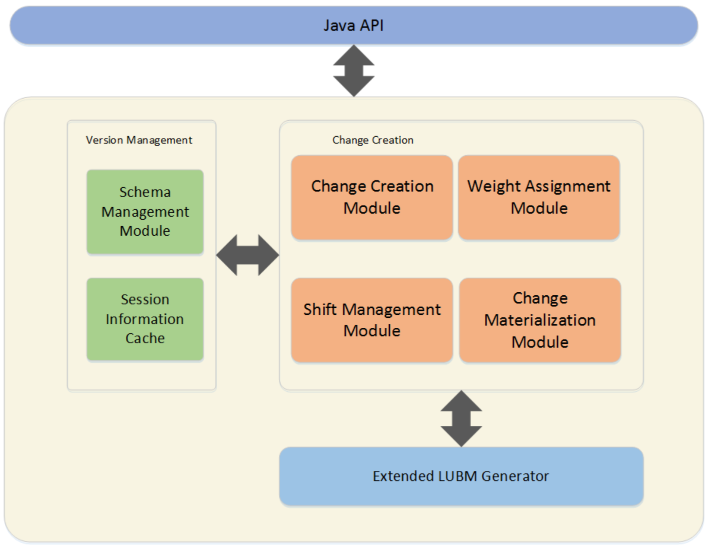

Fig. 2: High-level architecture of *EvoGen*.

context of the generation. The process also involves several degrees of randomization with respect to URI and literal values, cardinalities of inter-class properties, selection of characteristic set permutations and so on. This component is responsible for all interactions with the *Extended LUBM Generator component*, which performs the actual serialization of dataset versions in the file system. In order to distribute the computed changes, we perform weighting to each class and derive concrete numbers for the instance cardinalities. This weighting is done in the *Weight Assignment Module*, which uses normalized weights in the range of 0..1 for each class, based on studying LUBM's original data structure and total instances per class for various input dataset sizes. By multiplying these weights with the desired shift value $h(D)|_{t_i}^{t_j}$, we end up with an approximation for the total number of instances per class.

The Change Materialization module is responsible for creating the change log file. It interacts with the Change Creation module sequentially, and creates an instance of the Change Ontology for each insertion and deletion of class instances.

The Version Management component keeps session information on each version during runtime, the schema of the dataset, the newly introduced classes and characteristic sets per version, the mapping of dataset versions to their respective files and folders in the file system and so on. Also, it is responsible for generating different types of archives, based on the user input; it can generate successive full materialized datasets without any change set produced, or change-based archives that includes an initial dataset with all successive deltas, or finally combinations of these approaches (hybrid storage).

6 Conclusions and Future Work

In this paper, we describe the latest version of *EvoGen*, a system for synthetic and evolving data generation, with instance and schema level capabilities. Furthermore, EvoGen provides custom workload generation, that creates queries based on the user's choice of query types and the context of the generated data.

As existing RDF benchmarks do not address dynamic data, i.e., data that change over time, we aim to bridge this gap with EvoGen, by providing a means to generate synthetic data with evolving entities, and evolving structure. For these reasons, we have defined and discussed several requirements and characteristics that concern the data generation, and we have implemented several of these characteristics within EvoGen.

As future work, we intend to extend the requirements presented herein, and address issues of scalability and efficiency, as well as provide thorough experimental evaluation of the system by using it to benchmark existing RDF versioning solutions.

Acknowledgements. This work is supported by the EU-funded ICT project *SlideWiki* (agreement no 688095).

References

1. S. Auer, T. Dalamagas, H. Parkinson, F. Bancilhon, G. Flouris, D. Sacharidis, P. Buneman, D. Kotzinos, Y. Stavrakas, V. Christophides, et al. Diachronic linked data: towards long-term preservation of structured interrelated information. In *Proceedings of the First International Workshop on Open Data*, pages 31–39. ACM, 2012.
2. A. Bernstein, M. Stocker, and C. Kiefer. Sparql query optimization using selectivity estimation. In *Poster Proceedings of the 6th International Semantic Web Conference (ISWC)*, 2007.
3. C. Bizer and A. Schultz. The berlin sparql benchmark, 2009.
4. M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an efficient rdf store over a relational database. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 121–132. ACM, 2013.
5. S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 145–156. ACM, 2011.
6. J. D. Fernández, A. Polleres, and J. Umbrich. Towards efficient archiving of dynamic linked open data. In *Proceedings of the 1st DIACHRON workshop*, 2015.
7. I. Foundoulaki and A. Kementsietsidis. Assessing the performance of rdf engines: Discussing rdf benchmarks. `http://www.ics.forth.gr/isl/RDF-Benchmarks-Tutorial/index.html`. Accessed: 2016-04-22.
8. J. Gray. *Benchmark Handbook: For Database and Transaction Processing Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
9. Y. Guo, Z. Pan, and J. Heflin. Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

10. M. F. Husain, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Data intensive query processing for large rdf graphs using cloud computing tools. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 1–10. IEEE, 2010.
11. E. G. Kalayci, T. E. Kalayci, and D. Birant. An ant colony optimisation approach for optimising sparql queries by reordering triple patterns. *Information Systems*, 50:51–68, 2015.
12. Z. Kaoudi, K. Kyzirakos, and M. Koubarakis. Sparql query optimization on top of dhts. In *The Semantic Web–ISWC 2010*, pages 418–435. Springer, 2010.
13. A. Letelier, J. Pérez, R. Pichler, and S. Skritek. Static analysis and optimization of semantic web queries. *ACM Transactions on Database Systems (TODS)*, 38(4):25, 2013.
14. M. Meimaris. Evogen: a generator for synthetic versioned rdf. In T. Palpanas, E. Pitoura, W. Martens, S. Maabout, and K. Stefanidis, editors, *Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference (EDBT/ICDT 2016) (EDBT/ICDT)*, number 1558 in CEUR Workshop Proceedings, Aachen, 2016.
15. M. Meimaris, G. Papastefanatos, and C. Pateritsas. An archiving system for managing evolution in the data web. In *Proceedings of the 1st DIACHRON workshop*, 2015.
16. M. Meimaris, G. Papastefanatos, C. Pateritsas, T. Galani, and Y. Stavrakas. Towards a framework for managing evolving information resources on the data web. In *PROFILES@ ESWC*, 2014.
17. M. Meimaris, G. Papastefanatos, S. Viglas, Y. Stavrakas, and C. Pateritsas. A query language for multi-version data web archives. *arXiv preprint arXiv:1504.01891*, 2015.
18. M. Morsey, J. Lehmann, S. Auer, and A. N. Ngomo. Dbpedia SPARQL benchmark - performance assessment with real queries on real data. In *10th International Semantic Web Conference - ISWC 2011, Bonn, Germany, October 23-27*, 2011.
19. T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for rdf queries with multiple joins. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 984–994. IEEE, 2011.
20. N. Papailiou, I. Konstantinou, D. Tsoumakos, and N. Koziris. H2rdf: adaptive query processing on rdf data in the cloud. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 397–400. ACM, 2012.
21. G. Papastefanatos. Challenges and opportunities in the evolving data web. In *Proceedings of the 1st International Workshop on Modeling and Management of Big Data (with ER 2013)*, pages 23–28, 2013.
22. V. Papavasileiou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. High-level change detection in rdf (s) kbs. *ACM Transactions on Database Systems (TODS)*, 38(1):1, 2013.
23. Y. Roussakis, I. Chrysakis, K. Stefanidis, G. Flouris, and Y. Stavrakas. A flexible framework for understanding the dynamics of evolving rdf datasets. In *The Semantic Web-ISWC 2015*, pages 495–512. Springer, 2015.
24. M. Schmidt, O. Görlitz, P. Haase, G. Ladwig, A. Schwarte, and T. Tran. Fedbench: A benchmark suite for federated semantic data query processing. In *The Semantic Web–ISWC 2011*, pages 585–600. Springer, 2011.
25. M. Schmidt, T. Hornung, G. Lausen, and C. Pinkel. Sp 2 bench: A sparql performance benchmark, icde. *Shanghai, China*, 2009.
26. M. Schmidt, M. Meier, and G. Lausen. Foundations of sparql query optimization. In *Proceedings of the 13th International Conference on Database Theory*, pages 4–33. ACM, 2010.

27. K. Stefanidis, I. Chrysakis, and G. Flouris. On designing archiving policies for evolving rdf datasets on the web. In *Conceptual Modeling*, pages 43–56. Springer, 2014.
28. M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer, and D. Reynolds. Sparql basic graph pattern optimization using selectivity estimation. In *Proceedings of the 17th international conference on World Wide Web*, pages 595–604. ACM, 2008.
29. M. Voigt, A. Mitschick, and J. Schulz. Yet another triple store benchmark? practical experiences with real-world data. In *SDA*, pages 85–94. Citeseer, 2012.